CHAPTER

15

Starting and Stopping Services

IN THIS CHAPTER

Understanding the various Linux init services

Auditing Linux daemon-controlled services

Stopping and starting services

Changing the Linux server's default runlevel

Removing services

The primary job of a Linux server system is to offer services to local or remote users. A server can provide access to web pages, files, database information, streaming music, or other types of content. Name servers can provide access to lists of host computer or usernames. Hundreds of these and other types of services can be configured on your Linux systems.

Ongoing services offered by a Linux system, such as access to a printer service or login service, are typically implemented by what is referred to as a *daemon* process. Most Linux systems have a method of managing each daemon process as a *service* using one of several popular initialization systems (also referred to as init systems). Advantages of using init systems include the ability to do the following:

Identify runlevels: Put together sets of services in what are referred to as *runlevels* or *targets*.

Establish dependencies: Set service dependencies so, for example, a service that requires network interfaces won't start until all network startup services have started successfully.

Set the default runlevel: Select which runlevel or target starts up when the system boots (a *default runlevel*).

Manage services: Run commands that tell individual services to start, stop, pause, restart, or even reload configuration files.

Several different init systems are in use with Linux systems today. The one you use depends on the Linux distribution and release that you are using. In this chapter, I cover the following init systems that have been used in Fedora, Red Hat Enterprise Linux, Ubuntu, and many other Linux distributions:

SysVinit: This traditional init system was created for UNIX System V systems in the early 1980s. It offers an easy-to-understand method of starting and stopping services based on runlevel. Most UNIX and Linux systems up until a few years ago used SysVinit.

Systemd: The latest versions of Fedora and RHEL use the `systemd` init system. It is the most complex of the init systems, but it also offers much more flexibility. `Systemd` offers not only features for starting and working with services, but also lets you manage sockets, devices, mount points, swap areas, and other unit types.

NOTE

If you are using an older version of Ubuntu, you probably used Upstart as your initialization system. Beginning with Ubuntu 15.04 (released April 28, 2015), Upstart was replaced by the `systemd` initialization daemon. Thus, Upstart will not be described in this book.

This chapter describes the `SysVinit` and `systemd` init systems. In the process of using the init system that matches your Linux distribution, you learn how the boot process works to start services, how you can start and stop services individually, and how you enable and disable services.

Understanding the Initialization Daemon (init or systemd)

In order to understand service management, you need to understand the initialization daemon. The initialization daemon can be thought of as the "mother of all processes." This daemon is the first process to be started by the kernel on the Linux server. For Linux distributions that use SysVinit, the init daemon is literally named `init`. For `systemd`, the init daemon is named `systemd`.

The Linux kernel has a process ID (PID) of 0. Thus, the initialization process (`init` or `systemd`) daemon has a parent process ID (PPID) of 0, and a PID of 1. Once started, `init` is responsible for spawning (launching) processes configured to be started at the server's boot time, such as the login shell (`getty` or `mingetty` process). It is also responsible for managing services.

The Linux `init` daemon was based on the UNIX System V `init` daemon. Thus, it is called the SysVinit daemon. However, it was not the only classic `init` daemon. The `init` daemon is not part of the Linux kernel. Therefore, it can come in different flavors, and Linux distributions can choose which flavor to use. Another classic `init` daemon was based on

Berkeley UNIX, also called BSD. Therefore, the two original Linux `init` daemons were BSD `init` and SysVinit.

The classic `init` daemons worked without problems for many years. However, these daemons were created to work within a static environment. As new hardware, such as USB devices, came along, the classic `init` daemons had trouble dealing with these and other hot-plug devices. Computer hardware had changed from static to event based. New `init` daemons were needed to deal with these fluid environments.

In addition, as new services came along, the classic `init` daemons had to deal with starting more and more services. Thus, the entire system initialization process was less efficient and ultimately slower.

The modern initialization daemons have tried to solve the problems of inefficient system boots and non-static environments. The most popular of the new initialization daemons is `systemd`. Ubuntu, RHEL, and Fedora distributions have made the move to the `systemd` daemon while maintaining backward compatibility to the classic SysVinit, Upstart, or BSD `init` daemons.

The `systemd` daemon, available from `http://docs.fedoraproject.org/en-US/quick-docs/understanding-and-administering-systemd`, was written primarily by Lennart Poettering, a Red Hat developer. It is currently used by all of the latest versions of Fedora, RHEL, OpenSUSE, and Ubuntu.

In order to manage your services properly, you need to know which initialization daemon your server has. Figuring that out can be a little tricky. The initialization process running on a SysVinit or Upstart is named `init`. For the first `systemd` systems, it was also called `init` but is now named `systemd`. Running `ps -e` can immediately tell you if yours is a `systemd` system:

```
# ps -e | head
  PID TTY          TIME CMD
    1 ?        00:04:36 systemd
    2 ?        00:00:03 kthreadd
    3 ?        00:00:15 ksoftirqd/0
```

If PID 1 is the `init` daemon for your system, try looking on the init *Wikipedia* page (`http://wikipedia.org/wiki/Init`) under "Other implementations." This will help you understand if your `init` daemon is SysVinit, Upstart, or some other initialization system.

Understanding the classic init daemons

The classic `init` daemons, SysVinit and BSD `init`, are worth understanding, even if your Linux server has a different `init` daemon. Not only is backward compatibility to the classics often used in the newer `init` daemons, but many are based upon them.

The classic SysVinit and BSD `init` daemons operate in a very similar fashion. Although in the beginning they may have been rather different, over time very few significant differences remained. For example, the older BSD `init` daemon would obtain configuration information from the /etc/ttytab file. Now, like the SysVinit daemon, the BSD init daemon's configuration information is taken at boot time from the /etc/inittab file. The following is a classic SysVinit /etc/inittab file:

```
# cat /etc/inittab
# inittab  This file describes how the INIT process should set up
# Default runlevel. The runlevels used by RHS are:
#   0 - halt (Do NOT set initdefault to this)
#   1 - Single user mode
#   2 - Multiuser, no NFS (Same as 3, if you do not have networking)
#   3 - Full multiuser mode
#   4 - unused
#   5 - X11
#   6 - reboot (Do NOT set initdefault to this)
#
id:5:initdefault:

# System initialization.
si::sysinit:/etc/rc.d/rc.sysinit

l0:0:wait:/etc/rc.d/rc 0
l1:1:wait:/etc/rc.d/rc 1
l2:2:wait:/etc/rc.d/rc 2
l3:3:wait:/etc/rc.d/rc 3
l4:4:wait:/etc/rc.d/rc 4
l5:5:wait:/etc/rc.d/rc 5
l6:6:wait:/etc/rc.d/rc 6

# Trap CTRL-ALT-DELETE
ca::ctrlaltdel:/sbin/shutdown -t3 -r now
pf::powerfail:/sbin/shutdown -f -h +2
"Power Failure; System Shutting Down"


# If power was restored before the shutdown kicked in, cancel it.
pr:12345:powerokwait:/sbin/shutdown -c
"Power Restored; Shutdown Cancelled"


# Run gettys in standard runlevels
1:2345:respawn:/sbin/mingetty tty1
2:2345:respawn:/sbin/mingetty tty2
3:2345:respawn:/sbin/mingetty tty3
4:2345:respawn:/sbin/mingetty tty4
5:2345:respawn:/sbin/mingetty tty5
6:2345:respawn:/sbin/mingetty tty6

# Run xdm in runlevel 5
x:5:respawn:/etc/X11/prefdm -nodaemon
```

TABLE 15.1 Standard Linux Runlevels

Runlevel #	Name	Description
0	Halt	All services are shut down, and the server is stopped.
1 or S	Single User Mode	The root account is automatically logged in to the server. Other users cannot log in to the server. Only the command-line interface is available. Network services are not started.
2	Multiuser Mode	Users can log in to the server, but only the command-line interface is available. On some systems, network interfaces and services are started; on others they are not. Originally, this runlevel was used to start dumb terminal devices so that users could log in (but no network services were started).
3	Extended Multiuser Mode	Users can log in to the server, but only the command-line interface is available. Network interfaces and services are started. This is a common runlevel for servers.
4	User Defined	Users can customize this runlevel.
5	Graphical Mode	Users can log in to the server. Command-line and graphical interfaces are available. Network services are started. This is a common runlevel for desktop systems.
6	Reboot	The server is rebooted.

The `/etc/inittab` file tells the `init` daemon which runlevel is the default runlevel. A *runlevel* is a categorization number that determines what services are started and what services are stopped. In the preceding example, a default runlevel of 5 is set with the line `id:5:initdefault:`. Table 15.1 shows the standard seven Linux runlevels.

Linux distributions can differ slightly on the definition of each runlevel as well as which runlevels are offered.

CAUTION

The only runlevels that should be used in the `/etc/inittab` file are 2 through 5. The other runlevels could cause problems. For example, if you put runlevel 6 in the `/etc/inittab` file as the default, when the server reboots, it would go into a loop and continue to reboot over and over again.

The runlevels are not only used as a default runlevel in the `/etc/inittab` file. They can also be called directly using the `init` daemon itself. Thus, if you want to halt your server immediately, you type `init 0` at the command line:

```
# init 0
...
System going down for system halt NOW!
```

The `init` command accepts any of the runlevel numbers in Table 15.1, allowing you to switch your server quickly from one runlevel category to another. For example, if you need to perform troubleshooting that requires the graphical interface to be down, you can type **init 3** at the command line:

```
# init 3
INIT: Sending processes the TERM signal
starting irqbalance:                     [ OK ]
Starting setroubleshootd:
Starting fuse:  Fuse filesystem already available.
...
Starting console mouse services:         [ OK ]
```

To see your Linux server's current runlevel, simply type in the command **runlevel**. The first item displayed is the server's previous runlevel, which in the following example is 5. The second item displayed shows the server's current runlevel, which in this example is 3.

```
$ runlevel
5 3
```

In addition to the init command, you can use the telinit command, which is functionally the same. In the example that follows, the telinit command is used to reboot the server by taking it to runlevel 6:

```
# telinit 6
INIT: Sending processes the TERM signal
Shutting down smartd:                          [ OK ]
Shutting down Avahi daemon:                    [ OK ]
Stopping dhcdbd:                               [ OK ]
Stopping HAL daemon:                           [ OK ]
...
Starting killall:
Sending all processes the TERM signal...       [ OK ]
Sending all processes the KILL signal...       [ OK ]
...
Unmounting filesystems                         [ OK ]
Please stand by while rebooting the system
...
```

On a freshly booted Linux server, the current runlevel number should be the same as the default runlevel number in the /etc/inittab file. However, notice that the previous runlevel in the example that follows is N. The N stands for "Nonexistent" and indicates that the server was freshly booted to the current runlevel.

```
$ runlevel
N 5
```

How does the server know which services to stop and which ones to start when a particular runlevel is chosen? When a runlevel is chosen, the scripts located in the /etc/rc.d/rc#.d directory (where # is the chosen runlevel) are run. These scripts are run whether the runlevel is chosen via a server boot and the /etc/inittab initdefault setting or the init or telinit command is used. For example, if runlevel 5 is chosen, then all of the scripts in the /etc/rc.d/rc5.d directory are run; your list will be different, depending on what services you have installed and enabled.

```
# **ls /etc/rc.d/rc5.d**
K01smolt                     K88wpa_supplicant   S22messagebus
K02avahi-dnsconfd            K89dund             S25bluetooth
K02NetworkManager            K89netplugd         S25fuse
K02NetworkManagerDispatcher  K89pand             S25netfs
K05saslauthd                 K89rdisc            S25pcscd
K10dc_server                 K91capi             S26hidd
K10psacct                    S00microcode_ctl    S26udev-post
K12dc_client                 S04readahead_early  S28autofs
K15gpm                       S05kudzu            S50hplip
K15httpd                     S06cpuspeed         S55cups
K20nfs                       S08ip6tables        S55sshd
K24irda                      S08iptables         S80sendmail
K25squid                     S09isdn             S90ConsoleKit
K30spamassassin              S10network          S90crond
K35vncserver                 S11auditd           S90xfs
K50netconsole                S12restorecond      S95anacron
K50tux                       S12syslog           S95atd
K69rpcsvcgssd                S13irqbalance       S96readahead_later
K73winbind                   S13mcstrans         S97dhcdbd
K73ypbind                    S13rpcbind          S97yum-updatesd
K74nscd                      S13setroubleshoot   S98avahi-daemon
K74ntpd                      S14nfslock          S98haldaemon
K84btseed                    S15mdmonitor        S99firstboot
K84bttrack                   S18rpcidmapd        S99local
K87multipathd                S19rpcgssd          S99smartd
```

Notice that some of the scripts within the /etc/rc.d/rc5.d directory start with a K and some start with an S. The K refers to a script that will kill (stop) a process. The S refers to a script that will start a process. Also, each K and S script has a number before the name of the service or daemon that they control. This allows the services to be stopped or started in a particular controlled order. You would not want your Linux server's network services to be started before the network itself was started.

An /etc/rc.d/rc#.d directory exists for all the standard Linux runlevels. Each one contains scripts to start and stop services for its particular runlevel.

```
# **ls -d /etc/rc.d/rc?.d**
/etc/rc.d/rc0.d  /etc/rc.d/rc2.d  /etc/rc.d/rc4.d  /etc/rc.d/rc6.d
/etc/rc.d/rc1.d  /etc/rc.d/rc3.d  /etc/rc.d/rc5.d
```

Actually, the files in the /etc/rc.d/rc#.d directories are not scripts but instead symbolic links to scripts in the /etc/rc.d/init.d directory. Thus, there is no need to have multiple copies of particular scripts.

```
# **ls -l /etc/rc.d/rc5.d/K15httpd**
lrwxrwxrwx 1 root root 15 Oct 10 08:15
```

```
 /etc/rc.d/rc5.d/K15httpd -> ../init.d/httpd
# ls /etc/rc.d/init.d
anacron            functions  multipathd               rpcidmapd
atd                fuse       netconsole               rpcsvcgssd
auditd             gpm        netfs                    saslauthd
autofs             haldaemon  netplugd                 sendmail
avahi-daemon       halt       network                  setroubleshoot
avahi-dnsconfd     hidd       NetworkManager           single
bluetooth          hplip      NetworkManagerDispatcher smartd
btseed             hsqldb     nfs                      smolt
bttrack            httpd      nfslock                  spamassassin
capi               ip6tables  nscd                     squid
ConsoleKit         iptables   ntpd                     sshd
cpuspeed           irda       pand                     syslog
crond              irqbalance pcscd                    tux
cups               isdn       psacct                   udev-post
cups-config-daemon killall    rdisc                    vncserver
dc_client          kudzu      readahead_early          winbind
dc_server          mcstrans   readahead_later          wpa_supplicant
dhcdbd             mdmonitor  restorecond              xfs
dund               messagebus rpcbind                  ypbind
firstboot          microcode  rpcgssd                  yum-updatesd
```

Notice that each service has a single script in `/etc/rc.d/init.d`. There aren't separate scripts for stopping and starting a service. These scripts will stop or start a service depending upon what parameter is passed to them by the `init` daemon.

Each script in `/etc/rc.d/init.d` takes care of all that is needed for starting or stopping a particular service on the server. The following is a partial example of the `httpd` script on a Linux system that uses the SysVinit daemon. It contains a case statement for handling the parameter (`$1`) that was passed to it, such as `start`, `stop`, `status`, and so on.

```
# cat /etc/rc.d/init.d/httpd
#!/bin/bash
#
# httpd        Startup script for the Apache HTTP Server
#
# chkconfig: - 85 15
# description: Apache is a World Wide Web server.
#              It is used to serve \
#              HTML files and CGI.
# processname: httpd
# config: /etc/httpd/conf/httpd.conf
# config: /etc/sysconfig/httpd
# pidfile: /var/run/httpd.pid

# Source function library.
. /etc/rc.d/init.d/functions
...
```

```
# See how we were called.
case "$1" in
  start)
        start
        ;;
  stop)
        stop
        ;;
  status)
        status $httpd
        RETVAL=$?
        ;;
...
esac

exit $RETVAL
```

After the runlevel scripts linked from the appropriate `/etc/rc.d/rc#.d` directory are executed, the SysVinit daemon's process spawning is complete. The final step the `init` process takes at this point is to do anything else indicated in the `/etc/inittab` file (such as spawn `mingetty` processes for virtual consoles and start the desktop interface, if you are in runlevel `5`).

Understanding systemd initialization

The `systemd` initialization daemon is the newer replacement for the SysVinit and the Upstart `init` daemons. This modern initialization daemon was introduced in Fedora 15 and RHEL 7, and it is still in use today. It is backward compatible with both SysVinit and Upstart. System initialization time is reduced by `systemd` because it can start services in a parallel manner.

Learning systemd basics

With the SysVinit daemon, services are stopped and started based upon runlevels. The `systemd` service is concerned with runlevels, but it implements them in a different way with what are called *target units*. Although the main job of `systemd` is to start and stop services, it can manage other types of things referred to as units. A *unit* is a group consisting of a name, type, and configuration file, and it is focused on a particular service or action. There are 12 `systemd` unit types:

- `automount`
- `device`
- `mount`
- `path`
- `service`
- `snapshot`
- `socket`

- target
- timer
- swap
- slice
- scope

The two primary systemd units with which you need to be concerned for dealing with services are service units and target units. A *service unit* is for managing daemons on your Linux server. A *target unit* is simply a group of other units.

The example that follows shows several systemd service units and target units. The service units have familiar daemon names, such as cups and sshd. Note that each service unit name ends with .service. The target units shown have names like sysinit. (sysinit is used for starting up services at system initialization.) The target unit names end with .target.

```
# systemctl list-units | grep .service
...
cups.service           loaded active running CUPS Printing Service
dbus.service           loaded active running D-Bus Message Bus
...
NetworkManager.service loaded active running Network Manager
prefdm.service         loaded active running Display Manager
remount-rootfs.service loaded active exited  Remount Root FS
rsyslog.service        loaded active running System Logging
...
sshd.service           loaded active running OpenSSH server daemon
systemd-logind.service loaded active running Login Service
...
# systemctl list-units | grep .target
basic.target           loaded active active  Basic System
cryptsetup.target      loaded active active  Encrypted Volumes
getty.target           loaded active active  Login Prompts
graphical.target       loaded active active  Graphical Interface
local-fs-pre.target    loaded active active  Local File Systems (Pre)
local-fs.target        loaded active active  Local File Systems
multi-user.target      loaded active active  Multi-User
network.target         loaded active active  Network
remote-fs.target       loaded active active  Remote File Systems
sockets.target         loaded active active  Sockets
sound.target           loaded active active  Sound Card
swap.target            loaded active active  Swap
sysinit.target         loaded active active  System Initialization
syslog.target          loaded active active  Syslog
```

The Linux system unit configuration files are located in the `/lib/systemd/system` and `/etc/systemd/system` directories. You could use the `ls` command to look through those directories, but the preferred method is to use an option on the `systemctl` command as follows:

```
# systemctl list-unit-files --type=service
UNIT FILE                                    STATE
...
cups.service                                 enabled
...
dbus.service                                 static
...
NetworkManager.service                       enabled
...
poweroff.service                             static
...
sshd.service                                 enabled
sssd.service                                 disabled
...
276 unit files listed.
```

The unit configuration files shown in the preceding code are all associated with a service unit. Configuration files for target units can be displayed via the following method:

```
# systemctl list-unit-files --type=target
UNIT FILE                  STATE
anaconda.target            static
basic.target               static
bluetooth.target           static
cryptsetup.target          static
ctrl-alt-del.target        disabled
default.target             enabled
...
shutdown.target            static
sigpwr.target              static
smartcard.target           static
sockets.target             static
sound.target               static
swap.target                static
sysinit.target             static
syslog.target              static
time-sync.target           static
umount.target              static
43 unit files listed.
```

Notice that both of the configuration units' file examples display units with a status of static, enabled, or disabled. The enabled status means that the unit is currently enabled. The disabled status means that the unit is currently disabled. The next status, static, is slightly confusing. It stands for "statically enabled," and it means that the unit is enabled by default and cannot be disabled, even by root.

The service unit configuration files contain lots of information, such as what other services must be started, when this service can be started, which environmental file to use, and so on. The following example shows the sshd daemon's unit configuration file:

```
# cat /lib/systemd/system/sshd.service
[Unit]
Description=OpenSSH server daemon
Documentation=man:sshd(8) man:sshd_config(5)
After=network.target sshd-keygen.target

[Service]
Type=notify
EnvironmentFile=-/etc/crypto-policies/back-ends/opensshserver.config
EnvironmentFile=-/etc/sysconfig/sshd
ExecStart=/usr/sbin/sshd -D $OPTIONS $CRYPTO_POLICY
ExecReload=/bin/kill -HUP $MAINPID
KillMode=process
Restart=on-failure
RestartSec=42s

[Install]
WantedBy=multi-user.target

[Install]
WantedBy=multi-user.target
```

This basic service unit configuration file has the following options:

Description: A free-form description (comment line) of the service.

Documentation: Lists man pages for the sshd daemon and configuration file.

After: Configures ordering. In other words, it lists which units should be activated before this service is started.

Environment File: The service's configuration files.

ExecStart: The command used to start this service.

ExecReload: The command used to reload this service.

WantedBy: The target unit to which this service belongs.

Notice that the target unit, multi-user.target, is used in the sshd service unit configuration file. The sshd service unit is wanted by the multi-user.target. In other words, when the multi-user.target unit is activated, the sshd service unit is started.

You can view the various units that a target unit will activate by using the following command:

```
# systemctl show --property "Wants" multi-user.target
Wants=irqbalance.service firewalld.service plymouth-quit.service
systemd-update-utmp-runlevel.service systemd-ask-password-wall.path...
(END) q
```

Unfortunately, the systemctl command does not format the output for this well. It literally runs off the right edge of the screen so you cannot see the full results. Also, you must enter **q** to return to the command prompt. To fix this problem, pipe the output through some formatting commands to produce a nice, alphabetically sorted display, as shown in the example that follows:

```
# systemctl show --property "Wants" multi-user.target \
     | fmt -10 | sed 's/Wants=//g' | sort
atd.service
auditd.service
avahi-daemon.service
chronyd.service
crond.service
...
```

This display shows all of the services and other units that will be activated (started), including sshd, when the multi-user.target unit is activated. Remember that a target unit is simply a grouping of other units, as shown in the preceding example. Also notice that the units in this group are not all service units. There are path units and other target units as well.

A target unit has both *Wants* and requirements, called *Requires*. A *Wants* means that all of the units listed are triggered to activate (start). If they fail or cannot be started, no problem—the target unit continues on its merry way. The preceding example is a display of Wants only.

A Requires is much more stringent than a Wants and potentially catastrophic. A *Requires* means that all of the units listed are triggered to activate (start). If they fail or cannot be started, the entire unit (group of units) is deactivated.

You can view the various units a target unit Requires (must activate or the unit will fail), using the command in the example that follows. Notice that the Requires output is much shorter than the Wants for the multi-user.target. Thus, no special formatting of the output is needed.

```
# systemctl show --property "Requires" multi-user.target
Requires=basic.target
```

The target units also have configuration files, as do the service units. The following example shows the contents of the multi-user.target configuration file.

```
# cat /lib/systemd/system/multi-user.target
#  This file is part of systemd.
#
...

[Unit]
Description=Multi-User
Documentation=man:systemd.special(7)
Requires=basic.target
```

```
Conflicts=rescue.service rescue.target
After=basic.target rescue.service rescue.target
AllowIsolate=yes
```

This basic target unit configuration file has the following options:

`Description`: This is just a free-form description of the target.

`Documentation`: Lists the appropriate systemd man page.

`Requires`: If this `multi-user.target` gets activated, the listed target unit is also activated. If the listed target unit is deactivated or fails, then `multi-user.target` is deactivated. If there are no After and Before options, then both `multi-user.target` and the listed target unit activate simultaneously.

`Conflicts`: This setting avoids conflicts in services. Starting `multi-user.target` stops the listed targets and services, and vice versa.

`After`: This setting configures ordering. In other words, it determines which units should be activated before starting this service.

`AllowIsolate`: This option is a Boolean setting of `yes` or `no`. If this option is set to `yes`, then this target unit, `multi-user.target`, is activated along with its dependencies and all others are deactivated.

To get more information on these configuration files and their options, enter **`man systemd.service`**, **`man systemd.target`**, and **`man systemd.unit`** at the command line.

For the Linux server using `systemd`, the boot process is easier to follow now that you understand `systemd` target units. At boot, `systemd` activates the `default.target` unit. This unit is aliased either to `multi-user.target` or `graphical.target`. Thus, depending upon the `alias` set, the services targeted by the target unit are started.

If you need more help understanding the `systemd` daemon, you can enter **`man -k systemd`** at the command line to get a listing of the various `systemd` utilities' documentation in the man pages.

Learning systemd's backward compatibility to SysVinit

The `systemd` daemon has maintained backward compatibility to the SysVinit daemon. This allows Linux distributions time to migrate slowly to `systemd`.

While runlevels are not truly part of `systemd`, the `systemd` infrastructure has been created to provide compatibility with the concept of runlevels. There are seven target unit configuration files specifically created for backward compatibility to SysVinit:

- `runlevel0.target`
- `runlevel1.target`
- `runlevel2.target`
- `runlevel3.target`
- `runlevel4.target`

- runlevel5.target
- runlevel6.target

As you probably have already figured out, there is a target unit configuration file for each of the seven classic SysVinit runlevels. These target unit configuration files are symbolically linked to target unit configuration files that most closely match the idea of the original runlevel. In the example that follows, the symbolic links are shown for runlevel target units. Notice that the runlevel target units for runlevel 2, 3, and 4 are all symbolically linked to multi-user.target. The multi-user.target unit is similar to the legacy extended multi-user mode.

```
# ls -l /lib/systemd/system/runlevel*.target
lrwxrwxrwx. 1 root root 15 Apr  9 04:25 /lib/systemd/system/runlevel0.target
    -> poweroff.target
lrwxrwxrwx. 1 root root 13 Apr  9 04:25 /lib/systemd/system/runlevel1.target
    -> rescue.target
lrwxrwxrwx. 1 root root 17 Apr  9 04:25 /lib/systemd/system/runlevel2.target
    -> multi-user.target
lrwxrwxrwx. 1 root root 17 Apr  9 04:25 /lib/systemd/system/runlevel3.target
    -> multi-user.target
lrwxrwxrwx. 1 root root 17 Apr  9 04:25 /lib/systemd/system/runlevel4.target
    -> multi-user.target
lrwxrwxrwx. 1 root root 16 Apr  9 04:25 /lib/systemd/system/runlevel5.target
    -> graphical.target
lrwxrwxrwx. 1 root root 13 Apr  9 04:25 /lib/systemd/system/runlevel6.target
    -> reboot.target
```

The /etc/inittab file still exists, but it contains only comments stating that this configuration file is not used, and it gives some basic systemd information. The /etc/inittab file no longer has any true functional use. This is an example of an /etc/inittab file on a Linux server that uses systemd.

```
# cat /etc/inittab
# inittab is no longer used.
#
# ADDING CONFIGURATION HERE WILL HAVE NO EFFECT ON YOUR SYSTEM.
#
# Ctrl-Alt-Delete is handled by
# /etc/systemd/system/ctrl-alt-del.target
#
# systemd uses 'targets' instead of runlevels.
# By default, there are two main targets:
#
# multi-user.target: analogous to runlevel 3
# graphical.target: analogous to runlevel 5
#
# To view current default target, run:
# systemctl get-default
#
```

```
# To set a default target, run:
# systemctl set-default TARGET.target
```

The /etc/inittab explains that if you want something similar to a classic 3 or 5 runlevel as your default runlevel, you need run systemctl default.target to set the runlevel target to the one you want. To check what default.target is currently symbolically linked to (or in legacy terms, to check the default runlevel), use the command shown here. You can see that on this Linux server, the default is to start up at legacy runlevel 3.

```
# ls -l /etc/systemd/system/default.target
lrwxrwxrwx. 1 root root 36 Mar 13 17:27
 /etc/systemd/system/default.target ->
   /lib/systemd/system/runlevel3.target
```

The capability to switch runlevels using the init or telinit command is still available. When issued, either of the commands is translated into a systemd target unit activation request. Therefore, typing **init 3** at the command line really issues the command systemctl isolate multi-user.target. Also, you can still use the runlevel command to determine the current legacy runlevel, but it is strongly discouraged.

The classic SysVinit /etc/inittab handled spawning the getty or mingetty processes. The systemd init handles this via the getty.target unit. The getty.target is activated by the multi-user.target unit. You can see how these two target units are linked by the following command:

```
# systemctl show --property "WantedBy" getty.target
WantedBy=multi-user.target
```

Now that you have a basic understanding of classic and modern init daemons, it's time to do some practical server administrator actions that involve the initialization daemon.

Checking the Status of Services

As a Linux administrator, you need to check the status of the services being offered on your server. For security reasons, you should disable and remove any unused system services discovered through the process. Most important for troubleshooting purposes, you need to be able to know quickly what should and should not be running on your Linux server.

Of course, knowing which initialization service is being used by your Linux server is the first piece of information to obtain. How to determine this was covered in the section "Understanding the Initialization Daemon" earlier in this chapter. The following sections are organized into subsections on the various initialization daemons.

Checking services for SysVinit systems

To see all of the services that are being offered by a Linux server using the classic SysVinit daemon, use the chkconfig command. The example that follows shows the services available on a classic SysVinit Linux server. Note that each runlevel (0–6) is shown for each service with a status of on or off. The status denotes whether a particular service is started (on) or not (off) for that runlevel.

```
# chkconfig --list
ConsoleKit      0:off  1:off  2:off  3:on   4:on   5:on   6:off
NetworkManager  0:off  1:off  2:off  3:off  4:off  5:off  6:off
...
crond           0:off  1:off  2:on   3:on   4:on   5:on   6:off
cups            0:off  1:off  2:on   3:on   4:on   5:on   6:off
...
sshd            0:off  1:off  2:on   3:on   4:on   5:on   6:off
syslog          0:off  1:off  2:on   3:on   4:on   5:on   6:off
tux             0:off  1:off  2:off  3:off  4:off  5:off  6:off
udev-post       0:off  1:off  2:off  3:on   4:on   5:on   6:off
vncserver       0:off  1:off  2:off  3:off  4:off  5:off  6:off
winbind         0:off  1:off  2:off  3:off  4:off  5:off  6:off
wpa_supplicant  0:off  1:off  2:off  3:off  4:off  5:off  6:off
xfs             0:off  1:off  2:on   3:on   4:on   5:on   6:off
ypbind          0:off  1:off  2:off  3:off  4:off  5:off  6:off
yum-updatesd    0:off  1:off  2:off  3:on   4:on   5:on   6:off
```

Some services in the example are never started, such as vncserver. Other services, such as the cups daemon, are started on runlevels 2 through 5.

Using the chkconfig command, you cannot tell if a service is currently running. To do that, you need to use the service command. To help isolate only those services that are currently running, the service command is piped into the grep command and then sorted, as follows:

```
# service --status-all | grep running... | sort
anacron (pid 2162) is running...
atd (pid 2172) is running...
auditd (pid 1653) is running...
automount (pid 1952) is running...
console-kit-daemon (pid 2046) is running...
crond (pid 2118) is running...
cupsd (pid 1988) is running...
...
sshd (pid 2002) is running...
syslogd (pid 1681) is running...
xfs (pid 2151) is running...
yum-updatesd (pid 2205) is running...
```

You can also use both the `chkconfig` and the `service` commands to view an individual service's settings. Using both commands in the example that follows, you can view the `cups` daemon's settings.

```
# chkconfig --list cups
cups            0:off   1:off   2:on    3:on    4:on    5:on    6:off
#
# service cups status
cupsd (pid 1988) is running...
```

You can see that the `cupsd` daemon is set to start on every runlevel but 0, 1, and 6, and from the `service` command, you can see that it is currently running. Also, the process ID (PID) number is given for the daemon.

To see all of the services that are being offered by a Linux server using `systemd`, use the following command:

```
# systemctl list-unit-files --type=service | grep -v disabled
UNIT FILE                                    STATE
abrt-ccpp.service                            enabled
abrt-oops.service                            enabled
abrt-vmcore.service                          enabled
abrtd.service                                enabled
alsa-restore.service                         static
alsa-store.service                           static
anaconda-shell@.service                      static
arp-ethers.service                           enabled
atd.service                                  enabled
auditd.service                               enabled
avahi-daemon.service                         enabled
bluetooth.service                            enabled
console-kit-log-system-restart.service       static
console-kit-log-system-start.service         static
console-kit-log-system-stop.service          static
crond.service                                enabled
cups.service                                 enabled
...
sshd-keygen.service                          enabled
sshd.service                                 enabled
system-setup-keyboard.service                enabled
...
134 unit files listed.
```

Remember that the three status possibilities for a `systemd` service are enabled, disabled, or static. There's no need to include disabled to see which services are set to be active, which is effectively accomplished by using the `-v` option on the `grep` command, as shown in the preceding example. The state of static is essentially enabled and thus should be included.

To see if a particular service is running, use the following command:

```
# **systemctl status cups.service**
cups.service - CUPS Scheduler
  Loaded: loaded (/lib/systemd/system/cups.service; enabled)
  Active: active (running) since Wed 2019-09-18 17:32:27 EDT; 3 days ago
     Docs: man:cupsd(8)
 Main PID: 874 (cupsd)
   Status: "Scheduler is running..."
    Tasks: 1 (limit: 12232)
   Memory: 3.1M
   CGroup: /system.slice/cups.service
           └─874 /usr/sbin/cupsd -l
```

The `systemctl` command can be used to show the status of one or more services. In the preceding example, the printing service was chosen. Notice that the name of the service is `cups.service`. A great deal of helpful information about the service is given here, such as the fact that it is enabled and active, its start time, and its process ID (PID) as well.

Now that you can check the status of services and determine some information about them, you need to know how to accomplish starting, stopping, and reloading the services on your Linux server.

Stopping and Starting Services

The tasks of starting, stopping, and restarting services typically refer to immediate needs—in other words, managing services without a server reboot. For example, if you want to stop a service temporarily, then you are in the right place. However, if you want to stop a service and not allow it to be restarted at server reboot, then you actually need to disable the service, which is covered in the section "Enabling Persistent Services" later in this chapter.

Stopping and starting SysVinit services

The primary command for stopping and starting SysVinit services is the `service` command. With the `service` command, the name of the service that you want to control comes second in the command line. The last option is what you want to do to the service: `stop`, `start`, `restart`, and so on. The following example shows how to stop the `cups` service. Notice that an `OK` is given, which lets you know that `cupsd` has been successfully stopped:

```
# service cups status
cupsd (pid 5857) is running...
# service cups stop
Stopping cups:        [  OK  ]
# service cups status
cupsd is stopped
```

To start a service, you simply use a start option instead of a stop option on the end of the service command, as follows:

```
# service cups start
Starting cups:          [  OK  ]
# service cups status
cupsd (pid 6860) is running...
```

To restart a SysVinit service, the restart option is used. This option stops the service and then immediately starts it again:

```
# service cups restart
Stopping cups:           [  OK  ]
Starting cups:           [  OK  ]
# service cups status
cupsd (pid 7955) is running...
```

When a service is already stopped, a restart generates a FAILED status on the attempt to stop it. However, as shown in the example that follows, the service is successfully started when a restart is attempted:

```
# service cups stop
Stopping cups:            [  OK  ]
# service cups restart
Stopping cups:            [FAILED]
Starting cups:            [  OK  ]
# service cups status
cupsd (pid 8236) is running...
```

Reloading a service is different from restarting a service. When you reload a service, the service itself is not stopped. Only the service's configuration files are loaded again. The following example shows how to reload the cups daemon:

```
# service cups status
cupsd (pid 8236) is running...
# service cups reload
Reloading cups:             [  OK  ]
# service cups status
cupsd (pid 8236) is running...
```

If a SysVinit service is stopped when you attempt to reload it, you get a FAILED status. This is shown in the following example:

```
# service cups status
cupsd is stopped
# service cups reload
Reloading cups: [FAILED]
Stopping and starting systemd services
```

For the systemd daemon, the systemctl command works for stopping, starting, reloading, and restarting services. The options to the systemctl command should look familiar.

Stopping a service with systemd

In the example that follows, the status of the cups daemon is checked and then stopped using the systemctl stop cups.service command:

```
# systemctl status cups.service
cups.service - CUPS Printing Service
    Loaded: loaded (/lib/systemd/system/cups.service; enabled)
    Active: active (running) since Mon, 20 Apr 2020 12:36:3...
  Main PID: 1315 (cupsd)
    CGroup: name=systemd:/system/cups.service
            1315 /usr/sbin/cupsd -f
# systemctl stop cups.service
# systemctl status cups.service
cups.service - CUPS Printing Service
    Loaded: loaded (/lib/systemd/system/cups.service; enabled)
    Active: inactive (dead) since Tue, 21 Apr 2020 04:43:4...
    Process: 1315 ExecStart=/usr/sbin/cupsd -f
 (code=exited, status=0/SUCCESS)
    CGroup: name=systemd:/system/cups.service
```

Notice that when the status is taken, after stopping the cups daemon, the service is inactive (dead) but still considered enabled. This means that the cups daemon is still started upon server boot.

Starting a service with systemd

Starting the cups daemon is just as easy as stopping it. The example that follows demonstrates this ease:

```
# systemctl start cups.service
# systemctl status cups.service
cups.service - CUPS Printing Service
    Loaded: loaded (/lib/systemd/system/cups.service; enabled)
    Active: active (running) since Tue, 21 Apr 2020 04:43:5...
  Main PID: 17003 (cupsd)
    CGroup: name=systemd:/system/cups.service
           └ 17003 /usr/sbin/cupsd -f
```

After the cups daemon is started, using systemctl with the status option shows the service is active (running). Also, its process ID (PID) number, 17003, is shown.

Restarting a service with systemd

Restarting a service means that a service is stopped and then started again. If the service was not currently running, restarting it simply starts the service.

```
# systemctl restart cups.service
# systemctl status cups.service
cups.service - CUPS Printing Service
   Loaded: loaded (/lib/systemd/system/cups.service; enabled)
   Active: active (running) since Tue, 21 Apr 2020 04:45:2...
```

```
  Main PID: 17015 (cupsd)
   CGroup: name=systemd:/system/cups.service
           └  17015 /usr/sbin/cupsd -f
```

You can also perform a conditional restart of a service using `systemctl`. A conditional restart only restarts a service if it is currently running. Any service in an inactive state is not started.

```
# systemctl status cups.service
cups.service - CUPS Printing Service
  Loaded: loaded (/lib/systemd/system/cups.service; enabled)
  Active: inactive (dead) since Tue, 21 Apr 2020 06:03:32...
 Process: 17108 ExecStart=/usr/sbin/cupsd -f
 (code=exited, status=0/SUCCESS)
  CGroup: name=systemd:/system/cups.service
# systemctl condrestart cups.service
# systemctl status cups.service
cups.service - CUPS Printing Service
  Loaded: loaded (/lib/systemd/system/cups.service; enabled)
  Active: inactive (dead) since Tue, 21 Apr 2020 06:03:32...
 Process: 17108 ExecStart=/usr/sbin/cupsd -f
 (code=exited, status=0/SUCCESS)
  CGroup: name=systemd:/system/cups.service
```

Notice in the example that the `cups` daemon was in an inactive state. When the conditional restart was issued, no error messages were generated! The `cups` daemon was not started because conditional restarts affect active services. Thus, it is always a good practice to check the status of a service after stopping, starting, conditionally restarting, and so on.

Reloading a service with systemd

Reloading a service is different from restarting a service. When you `reload` a service, the service itself is not stopped. Only the service's configuration files are loaded again. Note that not all services are implemented to use the reload feature.

```
# systemctl status sshd.service
sshd.service - OpenSSH server daemon
   Loaded: loaded (/usr/lib/systemd/system/sshd.service; enabled)
   Active: active (running) since Wed 2019-09-18 17:32:27 EDT; 3 days ago
 Main PID: 1675 (sshd)
   CGroup: /system.slice/sshd.service
           └─1675 /usr/sbin/sshd -D
# systemctl reload sshd.service
# systemctl status sshd.service
sshd.service - OpenSSH server daemon
   Loaded: loaded (/lib/systemd/system/sshd.service; enabled)
   Active: active (running) since Wed 2019-09-18 17:32:27 EDT; 3 days ago
  Process: 21770 ExecReload=/bin/kill -HUP $MAINPID (code=exited, status=0/SUCCESS)
      (code=exited, status=0/SUCCESSd)
 Main PID: 1675 (sshd)
   CGroup: /system.slice/sshd.service
           └─1675 /usr/sbin/sshd -D ...
```

Doing a `reload` of a service, instead of a `restart` prevents any pending service operations from being aborted. A `reload` is a better method for a busy Linux server.

Now that you know how to stop and start services for troubleshooting and emergency purposes, you can learn how to enable and disable services.

Enabling Persistent Services

You use `stop` and `start` for immediate needs, not for services that need to be persistent. A *persistent service* is one that is started at server boot time or at a particular runlevel. Services that need to be set as persistent are typically new services that the Linux server is offering.

Configuring persistent services for SysVinit

One of the nice features of the classic SysVinit daemon is that making a particular service persistent or removing its persistence is very easy to do. Consider the following example:

```
# chkconfig --list cups
cups            0:off  1:off  2:off  3:off  4:off  5:off  6:off
```

On this Linux server, the `cups` service is not started at any runlevel, as shown with the `chkconfig` command. You can also check and see if any start (S) symbol links are set up in each of the seven runlevel directories, `/etc/rc.d/rc?.d`. Remember that SysVinit keeps symbolic links here for starting and stopping various services at certain runlevels. Each directory represents a particular runlevel; for example, `rc5.d` is for runlevel 5. Notice that only files starting with a `K` are listed, so there are links for killing off the `cups` daemon. None are listed with `S`, which is consistent with `chkconfig` because the `cups` daemon does not start at any runlevel on this server.

```
# ls /etc/rc.d/rc?.d/*cups
/etc/rc.d/rc0.d/K10cups  /etc/rc.d/rc3.d/K10cups
/etc/rc.d/rc1.d/K10cups  /etc/rc.d/rc4.d/K10cups
/etc/rc.d/rc2.d/K10cups  /etc/rc.d/rc5.d/K10cups
/etc/rc.d/rc6.d/K10cups
```

To make a service persistent at a particular runlevel, the `chkconfig` command is used again. Instead of the `--list` option, the `--level` option is used, as shown in the following code:

```
# chkconfig --level 3 cups on
# chkconfig --list cups
cups            0:off  1:off  2:off  3:on   4:off  5:off  6:off
# ls /etc/rc.d/rc3.d/S*cups
/etc/rc.d/rc3.d/S56cups
```

The service's persistence at runlevel 3 is verified by using both the chkconfig --list command and looking at the rc3.d directory for any files starting with the letter *S*.

To make a service persistent on more than one runlevel, you can do the following:

```
# chkconfig --level 2345 cups on
# chkconfig --list cups
cups            0:off  1:off  2:on   3:on   4:on   5:on   6:off
# ls /etc/rc.d/rc?.d/S*cups
/etc/rc.d/rc2.d/S56cups  /etc/rc.d/rc4.d/S56cups
/etc/rc.d/rc3.d/S56cups  /etc/rc.d/rc5.d/S56cups
```

Disabling a service is just as easy as enabling one with SysVinit. You just need to change the on in the chkconfig command to off. The following example demonstrates using the chkconfig command to disable the cups service at runlevel 5:

```
# chkconfig --level 5 cups off
# chkconfig --list cups
cups            0:off  1:off  2:on   3:on   4:on   5:off   6:off
# ls /etc/rc.d/rc5.d/S*cups
ls: cannot access /etc/rc.d/rc5.d/S*cups: No such file or directory
```

As expected, there is now no symbolic link, starting with the letter *S*, for the cups service in the /etc/rc.d/rc5.d directory.

For the systemd daemon, again the systemctl command is used. With it, you can disable and enable services on the Linux server.

Enabling a service with systemd

Using the enable option on the systemctl command sets a service to always start at boot (be persistent). The following shows exactly how to accomplish this:

```
# systemctl status cups.service
cups.service - CUPS Printing Service
   Loaded: loaded (/lib/systemd/system/cups.service; disabled)
   Active: inactive (dead) since Tue, 21 Apr 2020 06:42:38 ...
 Main PID: 17172 (code=exited, status=0/SUCCESS)
   CGroup: name=systemd:/system/cups.service
# systemctl enable cups.service
Created symlink /etc/systemd/system/printer.target.wants/cups.service
    → /usr/lib/systemd/system/cups.service.
Created symlink /etc/systemd/system/sockets.target.wants/cups.socket
    → /usr/lib/systemd/system/cups.socket.
Created symlink /etc/systemd/system/multi-user.target.wants/cups.path
    → /usr/lib/systemd/system/cups.path.
# systemctl status cups.service
cups.service - CUPS Printing Service
   Loaded: loaded (/lib/systemd/system/cups.service; enabled)
   Active: inactive (dead) since Tue, 21 Apr 2020 06:42:38...
 Main PID: 17172 (code=exited, status=0/SUCCESS)
   CGroup: name=systemd:/system/cups.service
```

Notice that the status of cups.service changes from disabled to enabled after using the enable option on systemctl. Also, notice that the enable option simply creates a few symbolic links. You may be tempted to create these links yourself. However, the preferred method is to use the systemctl command to accomplish this.

Disabling a service with systemd

You can use the disable option on the systemctl command to keep a service from starting at boot. However, it does not immediately stop the service. You need to use the stop option discussed in the section "Stopping a service with systemd." The following example shows how to disable a currently enabled service:

```
# systemctl disable cups.service
rm '/etc/systemd/system/printer.target.wants/cups.service'
rm '/etc/systemd/system/sockets.target.wants/cups.socket'
rm '/etc/systemd/system/multi-user.target.wants/cups.path'
# systemctl status cups.service
cups.service - CUPS Printing Service
   Loaded: loaded (/lib/systemd/system/cups.service; disabled)
   Active: active (running) since Tue, 21 Apr 2020 06:06:41...
 Main PID: 17172 (cupsd)
   CGroup: name=systemd:/system/cups.service
            17172 /usr/sbin/cupsd -f
```

The disable option simply removes a few files via the preferred method of the systemctl command. Notice also in the preceding example that although the cups service is now disabled, the cups daemon is still active (running) and needs to be stopped manually. With systemd, some services cannot be disabled. These services are static services. Consider the following service, dbus.service:

```
# systemctl status dbus.service
dbus.service - D-Bus System Message Bus
  Loaded: loaded (/lib/systemd/system/dbus.service; static)
  Active: active (running) since Mon, 20 Apr 2020 12:35:...
 Main PID: 707 (dbus-daemon)
...
# systemctl disable dbus.service
# systemctl status dbus.service
dbus.service - D-Bus System Message Bus
  Loaded: loaded (/lib/systemd/system/dbus.service; static)
  Active: active (running) since Mon, 20 Apr 2020 12:35:...
 Main PID: 707 (dbus-daemon)
...
```

When the systemctl disable command is issued on dbus.service, it is simply ignored. Remember that static means that the service is enabled by default and cannot be disabled, even by root. Sometimes, disabling a service is not enough to make sure that it does not run. For example, you might want network.service to replace NetworkManager.service for starting network interfaces on your system. Disabling NetworkManager would keep the service from starting on its own. However, if some other service listed NetworkManager as a dependency, that service would try to start NetworkManager when it started.

To disable a service in a way that prevents it from ever running on your system, you can use the `mask` option. For example, to set the NetworkManager service so that it never runs, type the following:

```
# systemctl mask NetworkManager.service
ln -s '/dev/null' '/etc/systemd/system/NetworkManager.service'
```

As the output shows, the `NetworkManager.service` file in `/etc` is linked to `/dev/null`. So even if someone tried to run that service, nothing would happen. To be able to use the service again, you could type `systemctl unmask NetworkManager.service`.

Now that you understand how to enable individual services to be persistent (and how to disable or mask individual services), you need to look at service groups as a whole. Next, I cover how to start groups of services at boot time.

Configuring a Default Runlevel or Target Unit

Whereas a persistent service is one that is started at server boot time, a persistent (default) runlevel or target unit is a group of services that are started at boot time. Both classic SysVinit and Upstart define these groups of services as runlevels, while `systemd` calls them target units.

Configuring the SysVinit default runlevel

You set the persistent runlevel for a Linux server using SysVinit in the `/etc/inittab` file. A portion of this file is shown here:

```
# cat /etc/inittab
#
# inittab       This file describes how the INIT process should
#               set up the system in a certain run-level.
...
id:5:initdefault:
...
```

The `initdefault` line in the example shows that the current default runlevel is runlevel 5. To change this, simply edit the `/etc/inittab` file using your favorite editor and change the 5 to one of the following runlevels: 2, 3, or 4. Do not use the runlevels 0 or 6 in this file! This would cause your server either to halt or reboot when it is started up.

For systemd, the term *target units* refers to groups of services to be started. The following shows the various target units that you can configure to be persistent and their equivalent backward-compatible, runlevel-specific target units:.

- multi-user.target =
  - runlevel2.target
  - runlevel3.target
  - runlevel4.target
- graphical.target = runlevel5.target

The persistent target unit is set via a symbolic link to the default.target unit file. Consider the following:

```
# ls -l /etc/systemd/system/default.target
lrwxrwxrwx. 1 root root 36 Mar 13 17:27
 /etc/systemd/system/default.target ->
 /lib/systemd/system/runlevel5.target
# ls -l /lib/systemd/system/runlevel5.target
lrwxrwxrwx. 1 root root 16 Mar 27 15:39
 /lib/systemd/system/runlevel5.target ->
 graphical.target
```

The example shows that the current persistent target unit on this server is runlevel5.target because default.target is a symbolic link to the runlevel5.target unit file. However, notice that runlevel5.target is also a symbolic link and it points to graphical.target. Thus, this server's current persistent target unit is graphical.target.

To set a different target unit to be persistent, you simply need to change the symbolic link for default.target. To be consistent, stick with the runlevel target units if they are used on your server.

The following systemctl example changes the server's persistent target unit from graphical.target to multi-user.target:

```
# systemctl get-default
graphical.target
#
 systemctl set-default runlevel3.target
 Removed /etc/systemd/system/default.target.
 Created symlink /etc/systemd/system/default.target →
/usr/lib/systemd/system/multi-user.target.
# systemctl get-default
 multi-user.target
```

When the server is rebooted, the multi-user.target is the persistent target unit. Any services in the multi-user.target unit are started (activated) at that time.

Adding New or Customized Services

Occasionally, you need to add a new service to your Linux server. Also, you may have to customize a particular service. When these needs arise, you must follow specific steps for your Linux server's initialization daemon to either take over the management of the service or recognize the customization of it.

Adding new services to SysVinit

When adding a new or customized service to a Linux SysVinit server, you must complete three steps in order to have the service managed by SysVinit:.

1. Create a new or customized service script file.
2. Move the new or customized service script to the proper location for SysVinit management.
3. Set appropriate permission on the script.
4. Add the service to a specific runlevel.

Step 1: Create a new or customized service script file

If you are customizing a service script, simply make a copy of the original unit file from `/etc/rc.d/init.d` and add any desired customizations.

If you are creating a new script, you need to make sure you handle all of the various options that you want the `service` command to accept for your service, such as `start`, `stop`, `restart`, and so on.

For a new script, especially if you have never created a service script before, it would be wise to make a copy of a current service script from `/etc/rc.d/init.d` and modify it to meet your new service's needs. Consider the following partial example of the `cupsd` service's script:

```
# cat /etc/rc.d/init.d/cups
#!/bin/sh
#
...
#   chkconfig: 2345 25 10

...
start () {
        echo -n $"Starting $prog: "
        # start daemon
        daemon $DAEMON
        RETVAL=$?
        echo
        [ $RETVAL = 0 ] && touch /var/lock/subsys/cups
        return $RETVAL
}
```

```
stop () {
        # stop daemon
        echo -n $"Stopping $prog: "
        killproc $DAEMON
        RETVAL=$?
       echo        [ $RETVAL = 0 ] && rm -f /var/lock/subsys/cups
}

restart() {
        stop
        start
}

case $1 in
...
```

The `cups` service script starts out by creating functions for each of the `start`, `stop`, and `restart` options. If you feel uncomfortable with shell script writing, review Chapter 7, "Writing Simple Shell Scripts," to improve your skills.

One line you should be sure to check and possibly modify in your new script is the `chkconfig` line that is commented out; for example:

```
#   chkconfig: 2345 25 10
```

When you add the service script in a later step, the `chkconfig` command reads that line to set runlevels at which the service starts (2, 3, 4, and 5), its run order when the script is set to start (25), and its kill order when it is set to stop (10).

Check the boot order in the default runlevel before adding your own script, as shown in this example:

```
# ls /etc/rc5.d
...
/etc/rc5.d/S22messagebus
/etc/rc5.d/S23NetworkManager
/etc/rc5.d/S24nfslock
/etc/rc5.d/S24openct
/etc/rc5.d/S24rpcgssd
/etc/rc5.d/S25blk-availability
/etc/rc5.d/S25cups
/etc/rc5.d/S25netfs
/etc/rc5.d/S26acpid
/etc/rc5.d/S26haldaemon
/etc/rc5.d/S26hypervkvpd
/etc/rc5.d/S26udev-post

...
```

In this case, the `chkconfig` line in the `S25My_New_Service` script will cause the script to be added after `S25cups` and before `S25netfs` in the boot order. You can change the `chkconfig` line in the service script if you want the service to start earlier (use a smaller number) or later (use a larger number) in the list of service scripts.

Step 2: Add the service script to /etc/rc.d/init.d

After you have modified or created and tested your service's script file, you can move it to the proper location, `/etc/rc.d/init.d`:

```
# cp My_New_Service /etc/rc.d/init.d
# ls /etc/rc.d/init.d/My_New_Service
/etc/rc.d/init.d/My_New_Service
```

Step 3: Set appropriate permission on the script

The script should be executable:

```
# chmod 755 /etc/rc.d/init.d/My_New_Service
```

Step 4: Add the service to runlevel directories

This final step sets up the service script to start and stop at different runlevels and checks that the service script works.

1. To add the script based on the `chkconfig` line in the service script, type the following:

   ```
   # chkconfig --add My_New_Service
   # ls /etc/rc?.d/*My_New_Service
   /etc/rc0.d/K10My_New_Service  /etc/rc4.d/S25My_New_Service
   /etc/rc1.d/K10My_New_Service  /etc/rc5.d/S25My_New_Service
   /etc/rc2.d/S25My_New_Service  /etc/rc6.d/K10My_New_Service
   /etc/rc3.d/S25My_New_Service
   ```

   Based on the previous example (`chkconfig: 2345 25 10`), symbolic links to the script set the service to start in the position 25 (`S25`) for runlevels 2, 3, 4, and 5. Also, links are set to stop (or not start) at runlevels 0, 1, and 6.

2. After you have made the symbolic link(s), test that your new or modified service works as expected before performing a server reboot.

   ```
   # service My_New_Service start
   Starting My_New_Service:       [  OK  ]
   # service My_New_Service stop
   ```

After everything is in place, your new or modified service starts at every runlevel that you have selected on your system. Also, you can start or stop it manually using the `service` command.

Adding new services to systemd

When adding a new or customized service to a Linux `systemd` server, you have to complete three steps in order to have the service managed by `systemd`:

1. Create a new or customized service configuration unit file for the new or customized service.
2. Move the new or customized service configuration unit file to the proper location for `systemd` management.
3. Add the service to a specific target unit's Wants to have the new or customized service start automatically with other services.

Step 1: Create a new or customized service configuration unit file

If you are customizing a service configuration unit file, simply make a copy of the original unit file from `/lib/systemd/system` and add any desired customizations.

For new files, obviously, you are creating a service unit configuration file from scratch. Consider the following basic service unit file template. At bare minimum, you need `Description` and `ExecStart` options for a service unit configuration file:

```
# cat My_New_Service.service
[Unit]
Description=My New Service
[Service]
ExecStart=/usr/bin/My_New_Service
```

For additional help on customizing or creating a new configuration unit file and the various needed options, you can use the man pages. At the command line, type **`man systemd.service`** to find out more about the various service unit file options.

Step 2: Move the service configuration unit file

Before you move the new or customized service configuration unit file, you need to be aware that there are two potential locations to store service configuration unit files. The one you choose determines whether the customizations take effect and if they remain persistent through software upgrades.

You can place your system service configuration unit file in one of the following two locations:

- `/etc/systemd/system`
  - This location is used to store customized local service configuration unit files.
  - Files in this location are not overwritten by software installations or upgrades.

    Files here are used by the system *even* if there is a file of the same name in the `/lib/systemd/system` directory.
- `/lib/systemd/system`
  - This location is used to store system service configuration unit files.
  - Files in this location are overwritten by software installations and upgrades.

Files here are used by the system only if there is no file of the same name in the /etc/systemd/system directory.

Thus, the best place to store your new or customized service configuration unit file is in /etc/systemd/system.

TIP

When you create a new or customized service, in order for the change to take effect without a server reboot, you need to issue a special command. At the command line, type **systemctl daemon-reload**.

Step 3: Add the service to the Wants directory

This final step is optional. It needs to be done only if you want your new service to start with a particular systemd target unit. For a service to be activated (started) by a particular target unit, it must be in that target unit's Wants directory.

First, add the line WantedBy=desired.target to the bottom of your service configuration unit file. The following example shows that the desired target unit for this new service is multi-user.target:

```
# cat /etc/systemd/system/My_New_Service.service
[Unit]
Description=My New Fake Service
[Service]
ExecStart=/usr/bin/My_New_Service
[Install]
WantedBy=multi-user.target
```

To add a new service unit to a target unit, you need to create a symbolic link. The following example shows the files located in the multi-user.target unit's Wants directory. Previously, in the section "Understanding systemd initialization," the systemctl command was used to list Wants, and it is still the preferred method. Notice that in this directory, the files are symbolic links pointing to service unit configuration files in the /lib/systemd/system directory.

```
# ls /etc/systemd/system/multi-user.target.wants
abrt-ccpp.service     cups.path           remote-fs.target
abrtd.service         fcoe.service        rsyslog.service
abrt-oops.service     irqbalance.service  sendmail.service
abrt-vmcore.service   lldpad.service      sm-client.service
atd.service           mcelog.service      sshd-keygen.service
auditd.service        mdmonitor.service   sshd.service
...
# ls -l /etc/systemd/system/multi-user.target.wants
total 0
lrwxrwxrwx. 1 root root 37 Nov  2 22:29 abrt-ccpp.service ->
    /lib/systemd/system/abrt-ccpp.service
lrwxrwxrwx. 1 root root 33 Nov  2 22:29 abrtd.service ->
    /lib/systemd/system/abrtd.service
...
```

```
lrwxrwxrwx. 1 root root 32 Apr 26 20:05 sshd.service ->
    /lib/systemd/system/sshd.service
```

The following illustrates the process of adding a symbolic link file for `My_New_Service`:

```
# ln -s /etc/systemd/system/My_New_Service.service
 /etc/systemd/system/multi-user.target.wants/My_New_Service.service
```

A symbolic link is created in the `multi-user.target.wants` directory. Now the new service, `My_New_Service`, is activated (started) when the `multi-user.target` unit is activated.

TIP

If you want to change the `systemd` target unit for a service, you need to change the symbol link to point to a new target `Wants` directory location. Use the `ln -sf` command to force any current symbolic link to be broken and the new designated symbolic link to be enforced.

Together, the three steps get your new or customized service added to a Linux `systemd` server. Remember that at this point, a new service is not running until a server reboot. To start the new service before a reboot, review the commands in the section "Stopping and Starting Services."

Summary

How you start and stop services is dependent upon what initialization daemon is used by your Linux server: SysVinit, Upstart, or Systemd. Before you do any service management, be sure to use the examples in this chapter to help you determine your Linux server's initialization daemon.

The concepts of starting and stopping services go along with other service management concepts, such as making a service persistent, starting certain services at server boot time, reloading a service, and restarting a service. Understanding these concepts is very helpful as you learn about configuring and managing a Linux print server in the next chapter.

Exercises

Refer to the material in this chapter to complete the tasks that follow. If you are stuck, solutions to the tasks are shown in Appendix B (although in Linux, there are often multiple ways to complete a task). Try each of the exercises before referring to the answers. These tasks assume that you are running a Fedora or Red Hat Enterprise Linux system (although some tasks work on other Linux systems as well).

1. Determine which initialization daemon your server is currently using.
2. What command can you use to check the status of the `sshd` daemon, depending on the initialization daemon in use on your Linux server?

3. Determine your server's previous and current runlevel.
4. How can you change the default runlevel or target unit on your Linux server?
5. For each initialization daemon, what commands list services running (or active) on your server?
6. List the running (or active) services on your Linux server.
7. For each initialization daemon, what commands show a particular service's current status?
8. Show the status of the `cups` daemon on your Linux server.
9. Attempt to restart the `cups` daemon on your Linux server.
10. Attempt to reload the `cups` daemon on your Linux server.